Artificial Intelligence project

Presented by :
Hagar Mahmoud, Fatma Sobhy,
Menna Ibrahim, Menna Mohammed

Maze Pathfinding problem

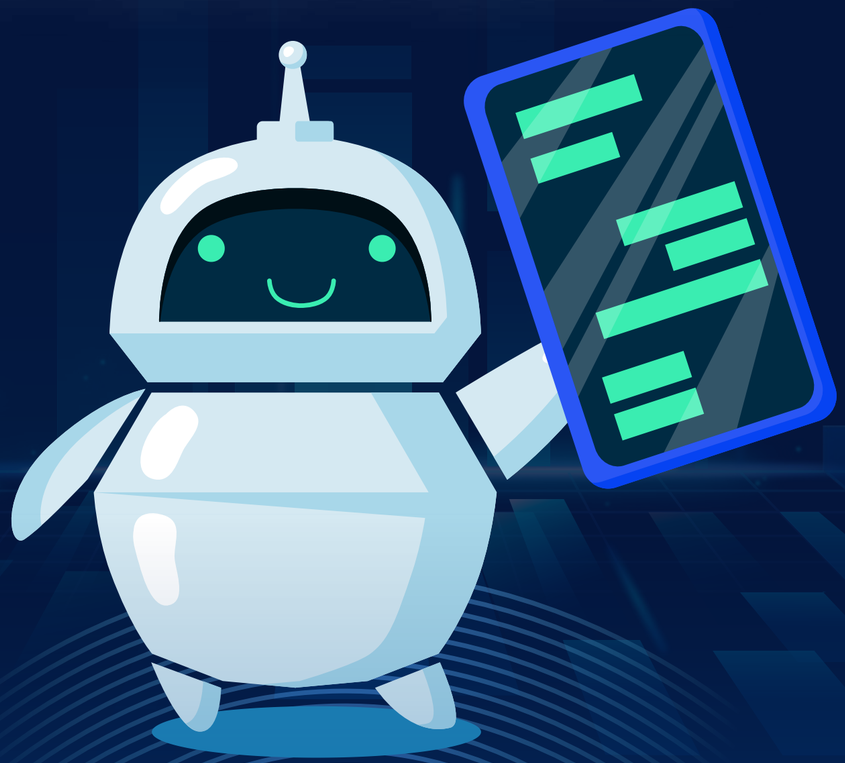
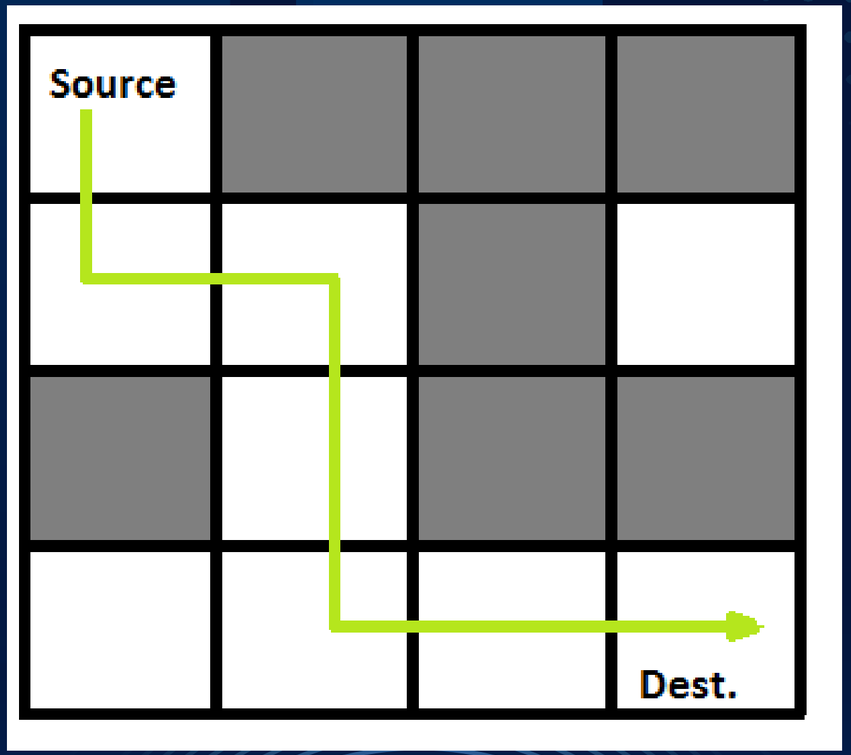


Table of contents

Maze Pathfinding problem

maze is generally represented as a 2D array or matrix consisting of rows and columns. Each cell is a intersection of a particular row and column and it represents a location in the grid. The cells in the grid may be open for traversal or may be blocked by an obstacle. If a cell is blocked by an obstacle that cell cannot be visited.

Path in a Grid or Maze refers to problems that involve navigating through a grid-like structure from the source (starting point) to the destination (endpoint) while avoiding the obstacles i.e., following rules and constraints.



Breadth-First Search (BFS)

Breadth-First Search (BFS)

01

Breadth-First Search (BFS) Overview:

BFS is a graph traversal algorithm that explores nodes level by level. Starting from the initial node, it first visits all neighboring nodes, then moves to the neighbors of those nodes, and continues this process until the goal is reached or all nodes are explored. BFS uses a queue to keep track of nodes to visit, ensuring that nodes are explored in the order they are discovered.

BFS Applied to Maze Solving:

When applied to a maze, BFS starts at the starting cell and explores all reachable cells step by step. Cells at distance 1 from the start are explored first, then cells at distance 2, and so on. This guarantees that the first time the goal cell is reached, the path found is one of the shortest possible in terms of steps.

02

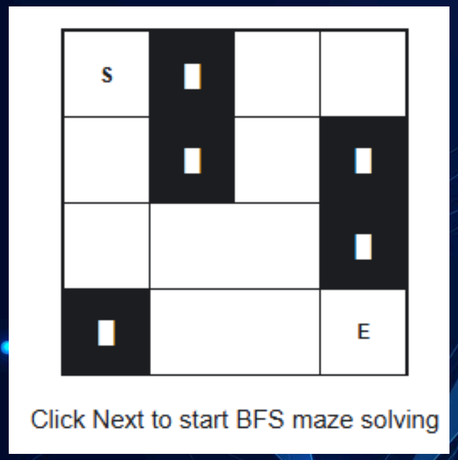
Think of BFS like ripples in a pond - it expands outward level by level from the starting point.
This guarantees that the first time we reach the destination, we've found the shortest path.

03

Time Complexity: O(n×m) where n and m are dimensions of the maze
Space Complexity: O(n×m) for the queue and visited array

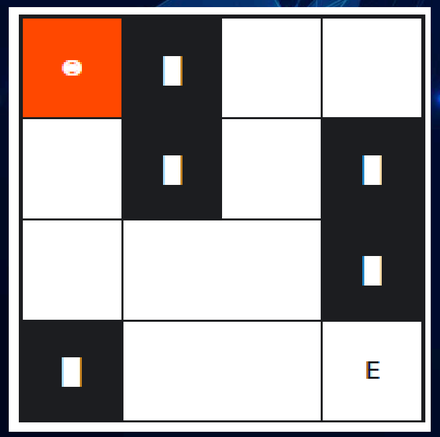
Example:

1



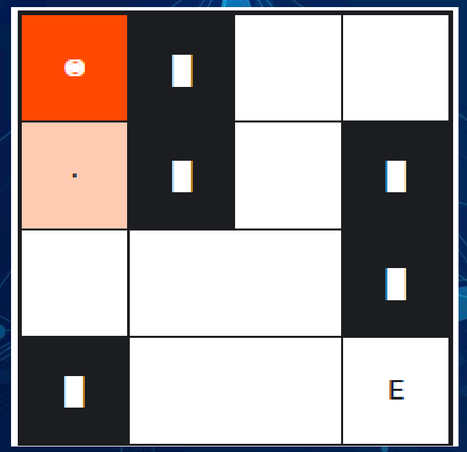
Starting BFS from (0,0)

2



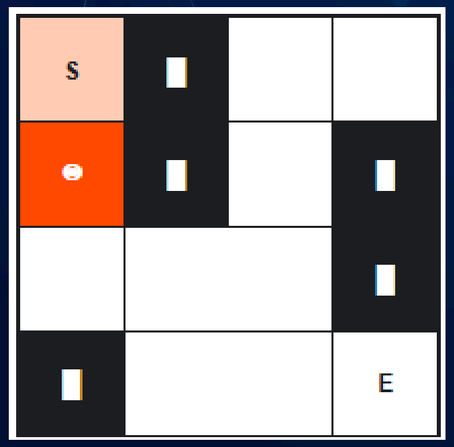
Processing cell (0, 0)

3



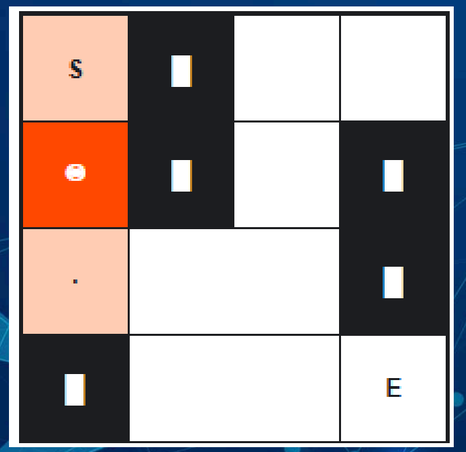
Processing cell (1, 0)

4



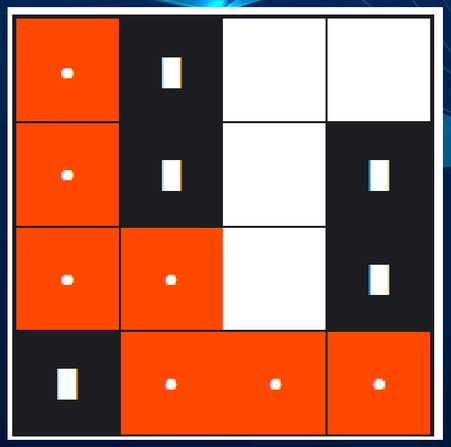
Added (2, 0) to queue.

5



Continue traversing.

6

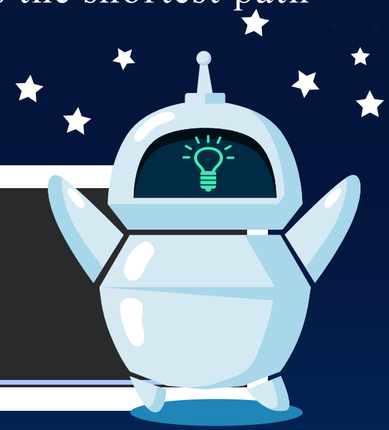


Implementation Code

```python
import random
from collections import deque

rows, cols = 50, 50
wall_prob = 0.3

random.seed(5)

maze = [[0 if random.random() > wall_prob else 1 for _ in range(cols)] for _ in range(rows)]

maze[0][0] = 0
maze[rows-1][cols-1] = 0

def bfs_maze(maze):
    rows, cols = len(maze), len(maze[0])
    directions = [(-1,0),(1,0),(0,-1),(0,1)]
    queue = deque([(0, 0, [(0, 0)])])
    visited = {(0, 0)}

    while queue:
        row, col, path = queue.popleft()
        if (row, col) == (rows-1, cols-1):
            return {
                "found": True,
                "path": path,
                "path_length": len(path),
                "nodes_explored": len(visited)
            }
        for dr, dc in directions:
            nr, nc = row + dr, col + dc
            if (0 <= nr < rows and 0 <= nc < cols and
                maze[nr][nc] == 0 and (nr, nc) not in visited):
                visited.add((nr, nc))
                queue.append((nr, nc, path + [(nr, nc)]))

    return {
        "found": False,
        "path": [],
        "path_length": 0,
        "nodes_explored": len(visited)
    }

result = bfs_maze(maze)

print("Path Found:", result["found"])
print("Path Length:", result["path_length"])
print("Nodes Explored:", result["nodes_explored"])
```

- The provided code demonstrates the implementation of the Breadth-First Search (BFS) algorithm to solve a maze pathfinding problem.
- It first generates a random maze of size 50x50 with walls and open cells, ensuring the start (top-left) and goal (bottom-right) cells are accessible.
- BFS starts from the initial cell (Start) and explores all neighboring cells level by level using a queue.
- Each visited cell is tracked in the visited set to avoid revisiting.
- Once the goal is reached, the algorithm outputs the final path from Start to Goal, the path length (number of steps), and the total number of nodes explored.
- This implementation clearly shows how BFS guarantees the shortest path while keeping track of exploration metrics

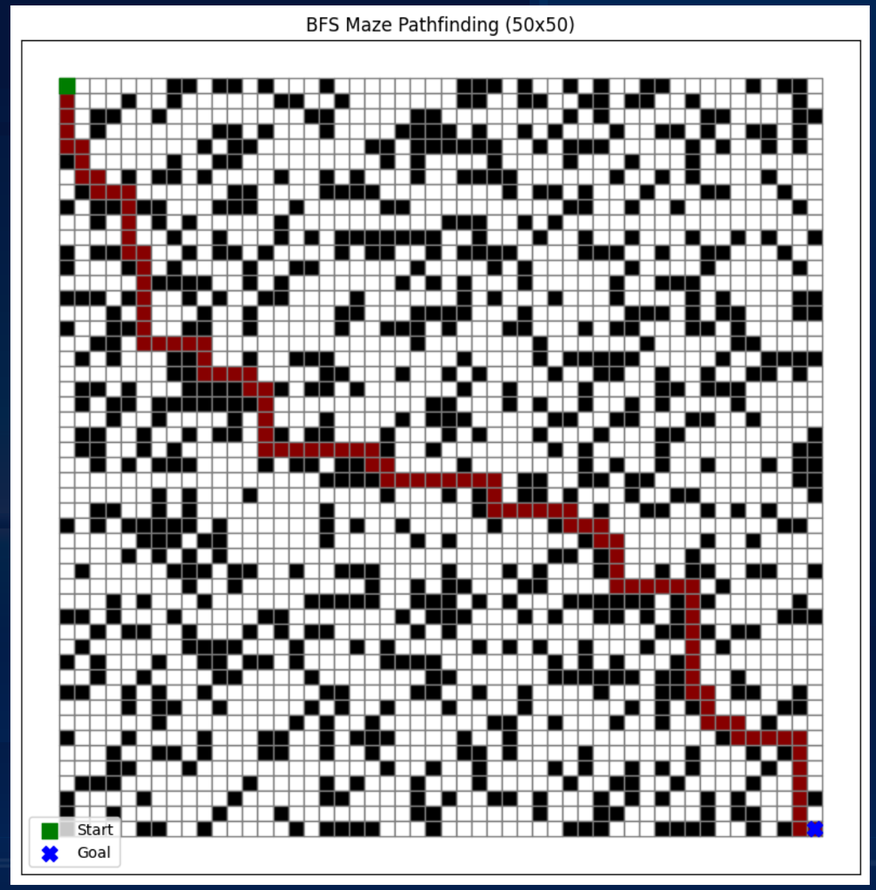
```
Path Found: True
Path Length: 99
Nodes Explored: 1679
```



Visualization Code

```python
import matplotlib.pyplot as plt

def visualize_maze(maze, path):
    rows, cols = len(maze), len(maze[0])
    plt.figure(figsize=(cols/5, rows/5))

    for r in range(rows):
        for c in range(cols):
            if maze[r][c] == 1:
                color = 'black'      # Walls
            elif (r, c) in path:
                color = 'darkred'    # Final Path
            else:
                color = 'white'      # Free cells

            plt.fill_between(
                [c, c+1],
                [rows-r-1]*2,
                [rows-r]*2,
                color=color,
                edgecolor='gray'
            )

    if path:
        sr, sc = path[0]
        gr, gc = path[-1]
        plt.scatter(sc+0.5, rows-sr-0.5, color='green', s=100, marker='s',
label='Start')
        plt.scatter(gc+0.5, rows-gr-0.5, color='blue', s=100, marker='X', label='Goal')

    plt.xticks([])
    plt.yticks([])
    plt.legend()
    plt.title("BFS Maze Pathfinding (50x50)")
    plt.show()

# Call visualization
if result["found"]:
    visualize_maze(maze, result["path"])
```

The visualization code simply draws the maze and highlights the BFS path. Walls are shown in black, free cells in white, the path in dark red, the start cell in green, and the goal cell in blue. A legend explains what each color represents, making it easy to see the shortest path found by BFS.



Greedy Best-First Search

Greedy Best-First Search

01

It’s an informed search algorithm that expands the node that is closest to the goal, as estimated by a heuristic function (explore the node that has the smallest value of h(n)).

02

Advantages

- Faster than uninformed search (like BFS or DFS) when a good heuristic is available.
- Memory efficient compared to algorithms like A* in some cases.

Disadvantages

- Not optimal — may find a longer path.
- Incomplete — can get stuck in loops if not tracking visited nodes.
- Heuristic dependent — poor heuristic can make it inefficient.

03

Time Complexity: O(n log n) where n is number of cells
Space Complexity: O(n)

Greedy Best-First Search

How it works?

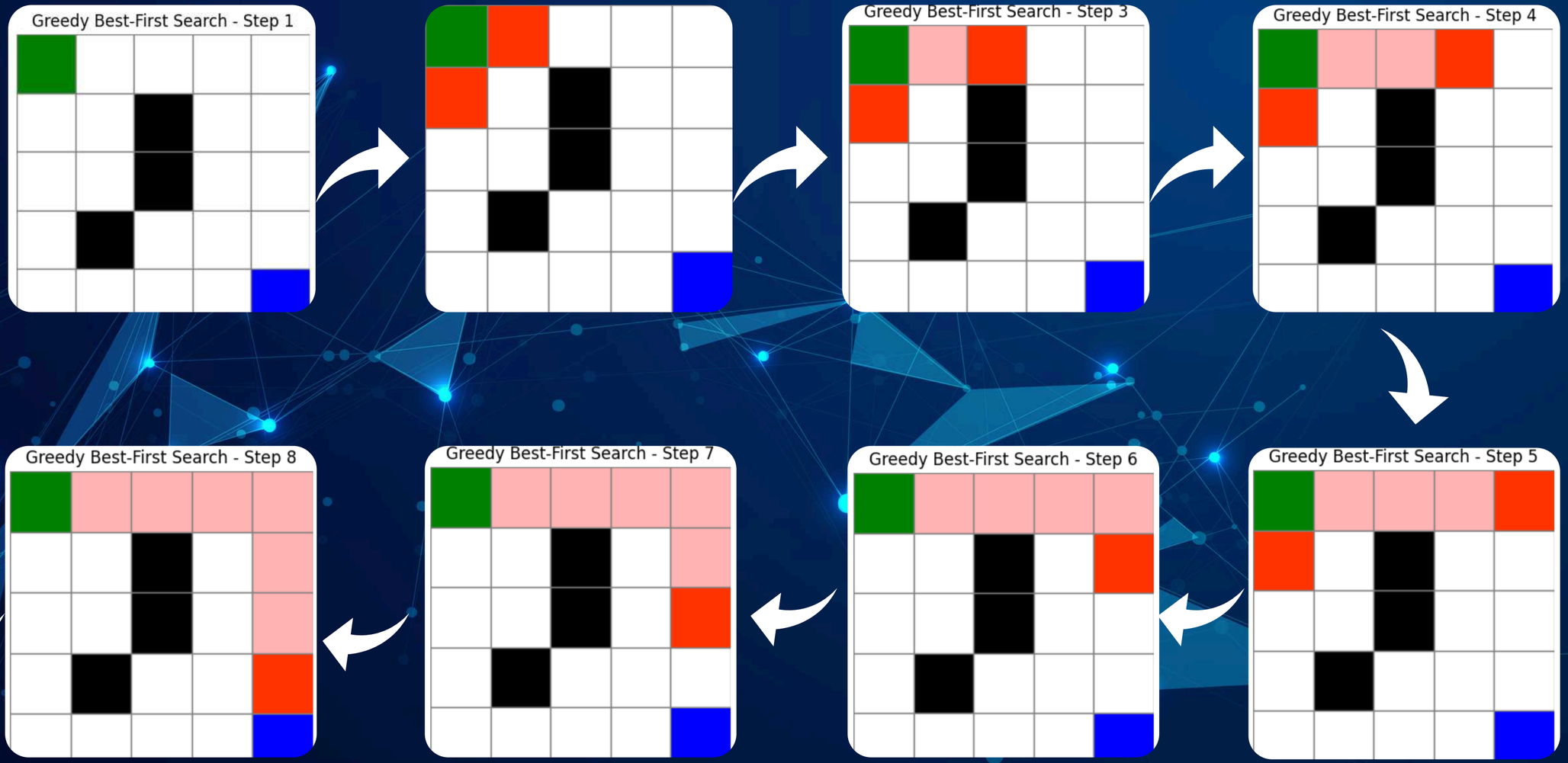
Start with an empty solution or initial state.

At each step, choose the locally best option according to a specific criterion.

Add the choice to the solution.

Repeat until the solution is complete or goal is reached.

Example:


Greedy Best-First Search - Step 1
Greedy Best-First Search - Step 3
Greedy Best-First Search - Step 4
Greedy Best-First Search - Step 5
Greedy Best-First Search - Step 6
Greedy Best-First Search - Step 7
Greedy Best-First Search - Step 8
Greedy Best-First Search - Step 10

Implementation Code

```python
import heapq
import random

def generate_maze(rows=50, cols=50, wall_prob=0.3, seed=123):
    random.seed(seed)
    maze = [[0 if random.random() > wall_prob else 1 for _ in range(cols)] for _ in range(rows)]
    maze[0][0] = 0
    maze[rows-1][cols-1] = 0
    return maze

def heuristic(pos, goal):

    return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1])

def greedy_maze(maze):
    rows, cols = len(maze), len(maze[0])
    start = (0, 0)
    goal = (rows-1, cols-1)
    directions = [(-1,0),(1,0),(0,-1),(0,1)]

    pq = [(heuristic(start, goal), start, [(0, 0)])]
    visited = {start}

    while pq:
        h_score, (row, col), path = heapq.heappop(pq)

        if (row, col) == goal:
            return {
                "found": True,
                "path": path,
                "path_length": len(path),
                "nodes_explored": len(visited)
            }

        for dr, dc in directions:
            nr, nc = row + dr, col + dc

            if (0 <= nr < rows and 0 <= nc < cols and maze[nr][nc] == 0 and (nr, nc) not in visited):
                visited.add((nr, nc))
                new_path = path + [(nr, nc)]
                heapq.heappush(pq, (heuristic((nr, nc), goal), (nr, nc), new_path))

    return {
        "found": False,
        "path": [],
        "path_length": 0,
        "nodes_explored": len(visited)
    }


maze = generate_maze(rows=50, cols=50, wall_prob=0.25, seed=123)
result = greedy_maze(maze)

print("Path Found:", result["found"])
print("Path Length:", result["path_length"])
print("Nodes Explored:", result["nodes_explored"])
```

The provided code implements the Greedy Best-First Search algorithm to solve a maze pathfinding problem.

A random 50×50 maze is generated with walls and open cells, ensuring that the start and goal cells are accessible.

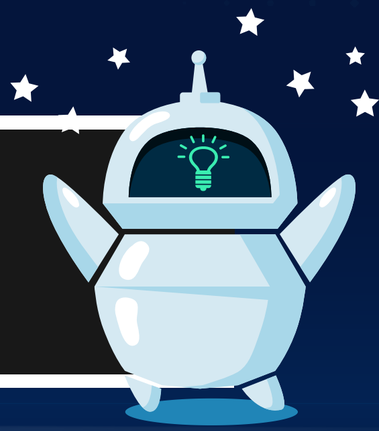
The algorithm starts from the initial cell and always selects the neighboring cell that is closest to the goal using a heuristic function.

A priority queue and a visited set are used to guide the search and avoid revisiting cells.

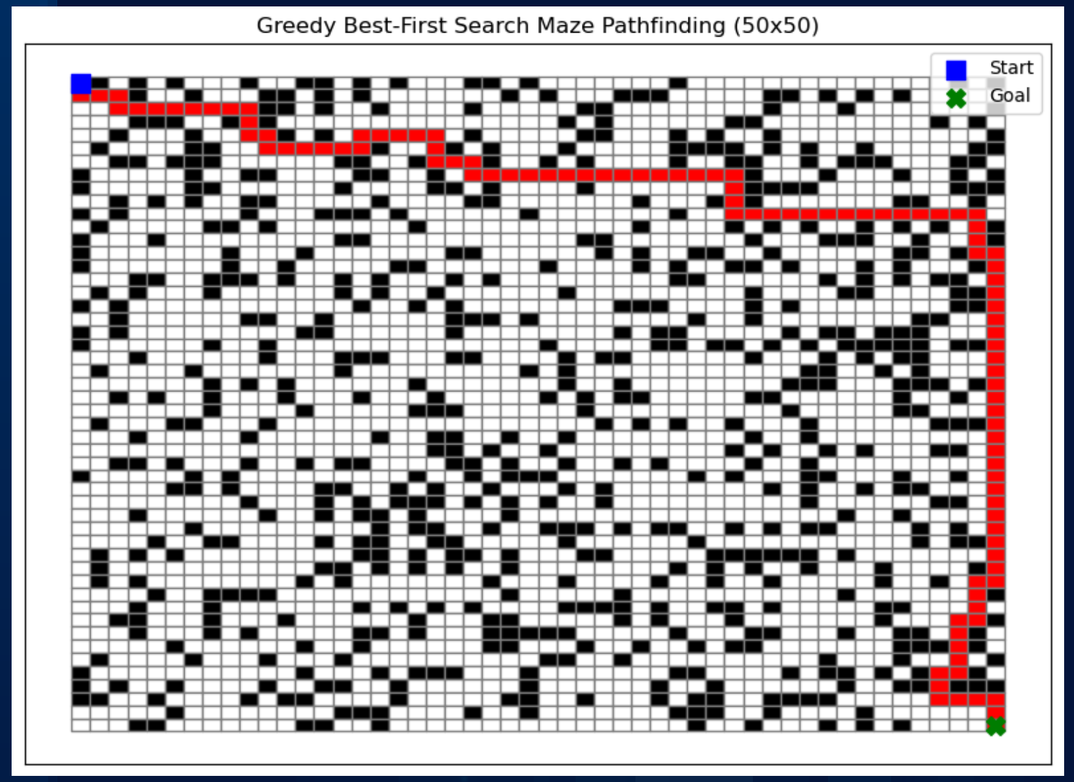
Once the goal is reached, the final path, path length, and number of explored nodes are reported.

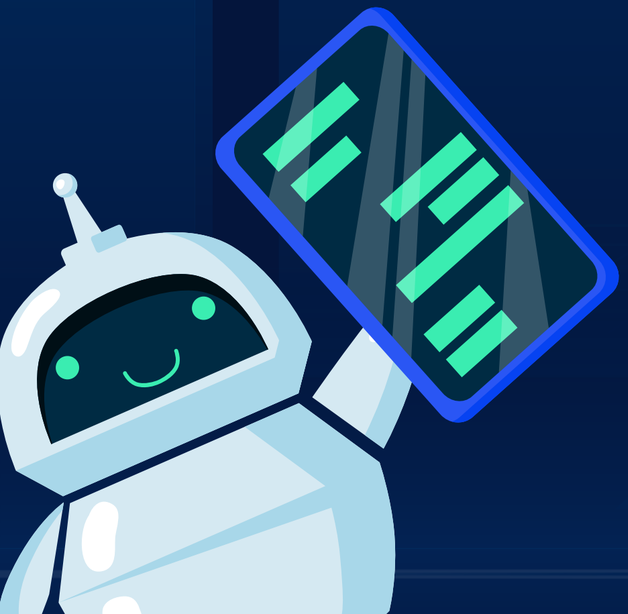
Greedy Best-First Search focuses on reaching the goal quickly

```
Path Found: True
Path Length: 107
Nodes Explored: 206
```



Visualization Code





The visualization code simply draws the maze and highlights the greedy path. Walls are shown in black, free cells in white, the path in dark red, the start cell in green, and the goal cell in blue. A legend explains what each color represents

```python
import matplotlib.pyplot as plt
import heapq
import random

def generate_maze(rows=50, cols=50, wall_prob=0.3, seed=123):

    random.seed(seed)
    maze = [[0 if random.random() > wall_prob else 1 for _ in range(cols)] for _ in range(rows)]
    maze[0][0] = 0
    maze[rows-1][cols-1] = 0
    return maze

def heuristic(position, goal):

    return abs(position[0] - goal[0]) + abs(position[1] - goal[1])

def greedy_search_maze(maze):
    if not maze or not maze[0] or maze[0][0] == 1:
        return False, []

    rows, cols = len(maze), len(maze[0])
    if maze[rows-1][cols-1] == 1:
        return False, []

    start = (0, 0)
    goal = (rows-1, cols-1)
    directions = [(-1,0),(1,0),(0,-1),(0,1)]

    pq = [(heuristic(start, goal), start, [(0, 0)])]
    visited = {start}

    while pq:
        f_score, (row, col), path = heapq.heappop(pq)

        if (row, col) == goal:
            return True, path

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc

            if (0 <= new_row < rows and 0 <= new_col < cols and
                maze[new_row][new_col] == 0 and (new_row, new_col) not in visited):

                visited.add((new_row, new_col))
                new_f_score = heuristic((new_row, new_col), goal)
                new_path = path + [(new_row, new_col)]
                heapq.heappush(pq, (new_f_score, (new_row, new_col), new_path))

    return False, []

def visualize_maze(maze, path):
    rows, cols = len(maze), len(maze[0])
    plt.figure(figsize=(cols/5, rows/5))

    for r in range(rows):
        for c in range(cols):
            if maze[r][c] == 1:
                color = 'black'
            elif (r,c) in path:
                color = 'red'
            else:
                color = 'white'
            plt.fill_between([c, c+1], [rows-r-1]*2, [rows-r]*2, color=color, edgecolor='gray')

    start_r, start_c = path[0]
    goal_r, goal_c = path[-1]
    plt.scatter(start_c+0.5, rows-start_r-0.5, color='blue', s=100, marker='s', label='Start')
    plt.scatter(goal_c+0.5, rows-goal_r-0.5, color='green', s=100, marker='X', label='Goal')

    plt.xticks([])
    plt.yticks([])
    plt.title("Greedy Best-First Search Maze Pathfinding (50×50)")
    plt.legend(loc='upper right')
    plt.show()


maze = generate_maze(50, 50, wall_prob=0.25, seed=123)
result, path = greedy_search_maze(maze)

print(f"Path exists: {result}")
if result:
    print(f"Path length: {len(path)} steps")
    visualize_maze(maze, path)
else:
    print("No path found in this maze!")
```

Depth-First Search (DFS)

Depth-First Search (DFS)

01

- **Depth-First Search (DFS)** is an uninformed search algorithm that explores paths by going as deep as possible before backtracking.
- Starting from the initial node, DFS explores one neighbor at a time, going as deep as possible along a path. When it reaches a dead end, it backtracks to the last node with unexplored neighbors and continues searching until the goal is found.
- DFS uses a stack (or recursion) to explore nodes deeply, backtracking when needed. It can find a path to the goal but doesn’t guarantee the shortest path.

02

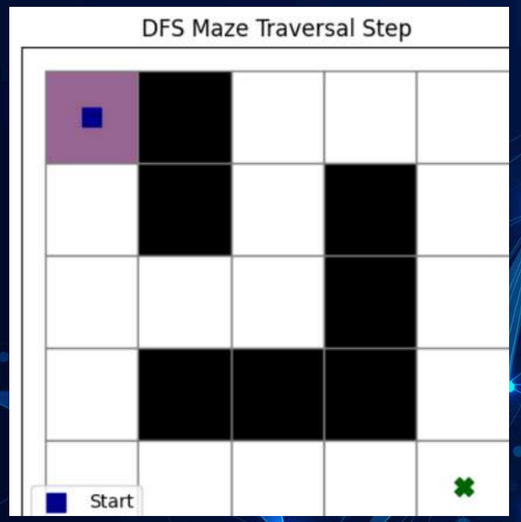
- When applied to a maze, DFS starts at the starting cell and explores one path as far as possible before backtracking. It does not guarantee the shortest path and may explore deep branches before finding the goal.

03

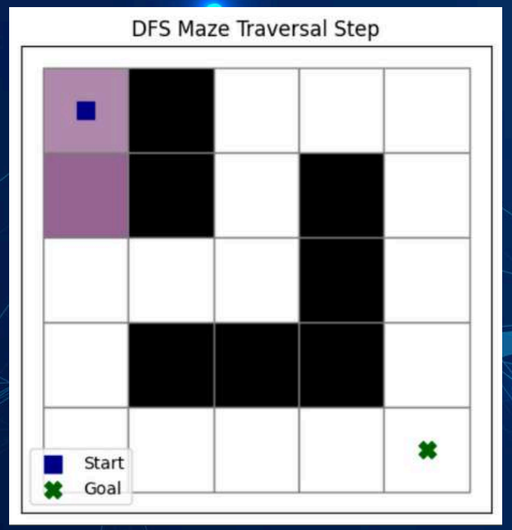
- Time Complexity: O(n×m)
- Space Complexity: O(n×m)

Example:

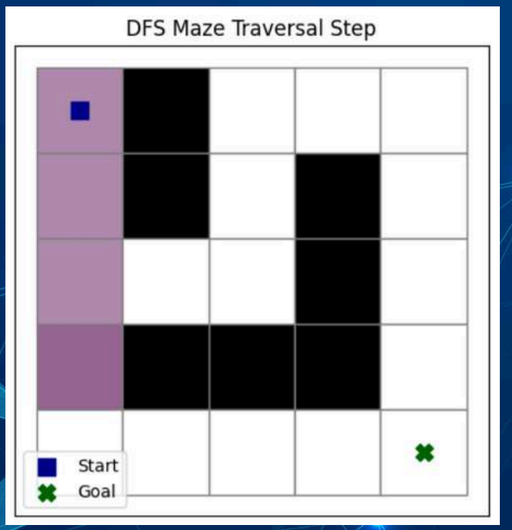
1



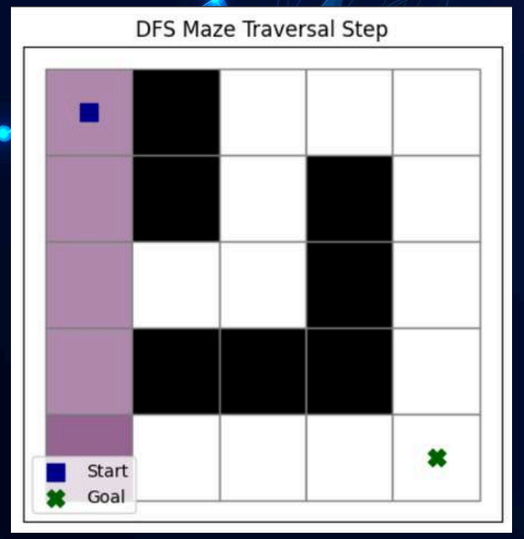
2



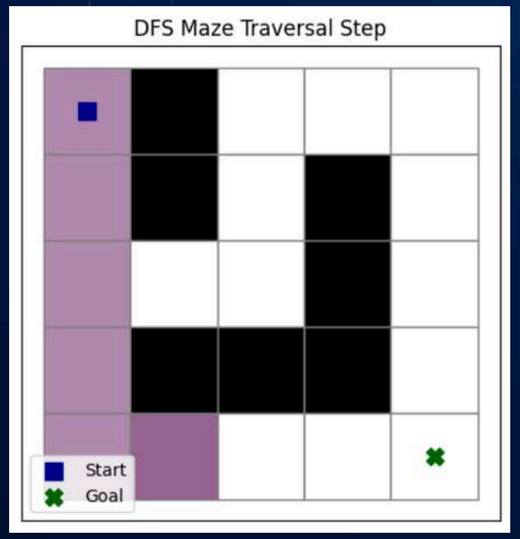
3



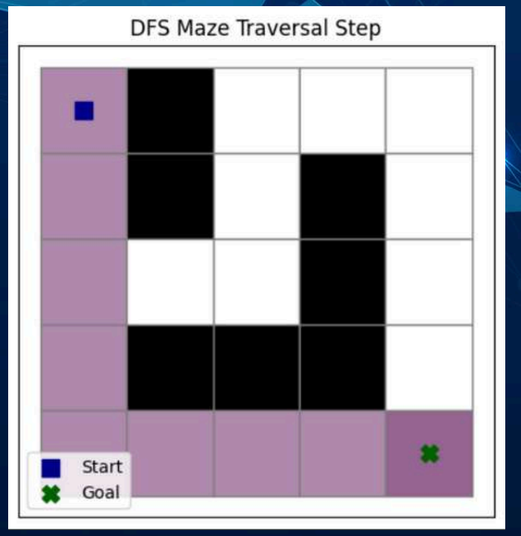
4



5



6



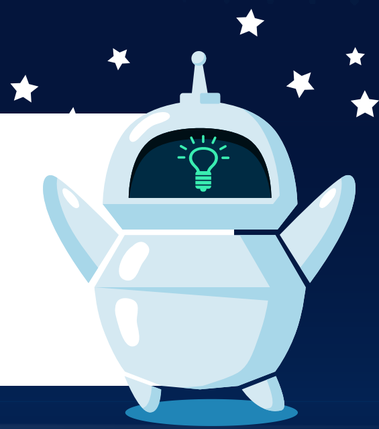
Implementation Code

```python
import numpy as np
import random

def generate_maze_with_path(rows=50, cols=50, seed=123):
    random.seed(seed)
    maze = np.ones((rows, cols), dtype=int)

    r = c = 0
    maze[r, c] = 0
    while r < rows-1 or c < cols-1:
        if r < rows-1 and c < cols-1:
            if random.choice([True, False]):
                r += 1
            else:
                c += 1
        elif r < rows-1:
            r += 1
        else:
            c += 1
        maze[r, c] = 0

    for i in range(rows):
        for j in range(cols):
            if maze[i, j] == 1 and random.random() > 0.4:
                maze[i, j] = 0

    return maze

def search_maze_dfs(maze):
    rows, cols = maze.shape
    stack = [(0, 0, [(0, 0)])]
    visited = {(0, 0)}
    directions = [(-1,0),(1,0),(0,-1),(0,1)]
    nodes_explored = 0

    while stack:
        row, col, path = stack.pop()
        nodes_explored += 1

        if (row, col) == (rows-1, cols-1):
            return {
                "path": path,
                "nodes_explored": nodes_explored
            }

        for dr, dc in directions:
            nr, nc = row + dr, col + dc
            if (0 <= nr < rows and 0 <= nc < cols and
                (nr, nc) not in visited and maze[nr, nc] == 0):
                stack.append((nr, nc, path + [(nr, nc)]))
                visited.add((nr, nc))

    return {
        "path": None,
        "nodes_explored": nodes_explored
    }

maze = generate_maze_with_path(50, 50, seed=123)
result = search_maze_dfs(maze)

path = result["path"]
print("Path exists:", path is not None)

if path:
    print("DFS path length:", len(path))
    print("Nodes explored:", result["nodes_explored"])
else:
    print("No path found")
```

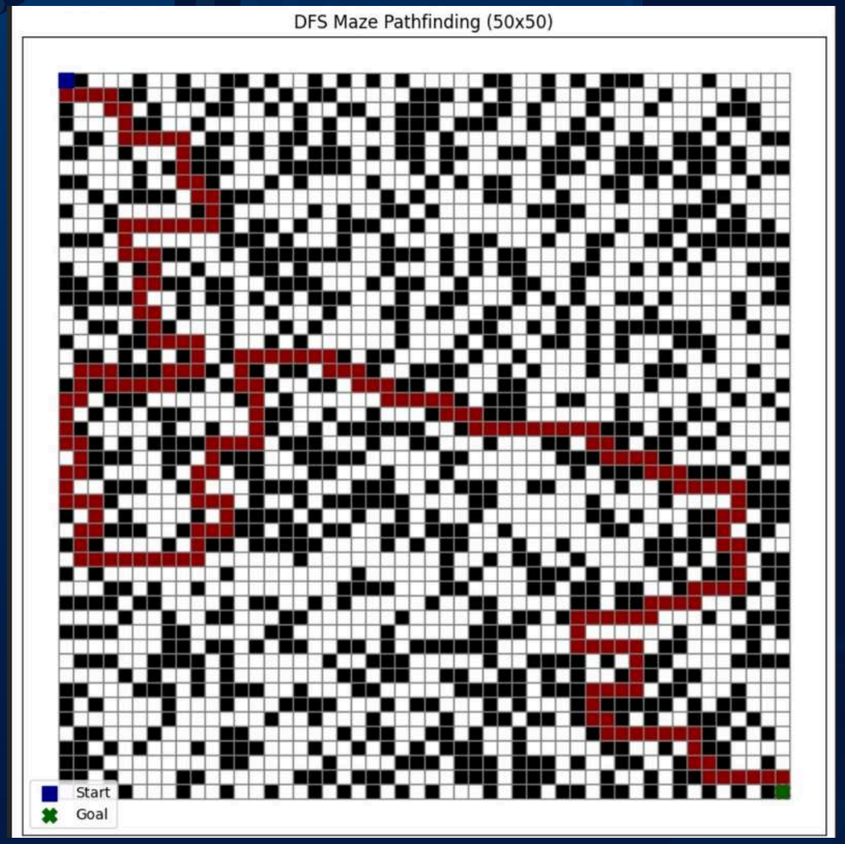
The provided code implements the Depth-First Search (DFS) algorithm to solve a maze pathfinding problem.

- problem.
- It generates a fixed 50×50 maze with walls and open cells, while ensuring that a valid path exists between the start cell (top-left) and the goal cell (bottom-right).
- DFS begins at the start cell and explores the maze by moving as deep as possible along one path before backtracking when no further moves are available.
- A stack is used to control the search order, and visited cells are stored to prevent revisiting the same positions.
- During the search, the algorithm tracks the path taken and counts the number of explored nodes. Once the goal is reached, it returns the found path, its length, and the total number of nodes explored.
- This implementation highlights how DFS focuses on deep exploration without guaranteeing the shortest path.

```
Path exists: True
DFS path length: 201
Nodes explored: 636
```



Visualization Code



The visualization code displays the maze and highlights the path found by the Depth-First Search (DFS) algorithm.

Walls are represented in black, free cells in white, and the path explored by DFS is shown in dark red.
The start cell is marked in dark blue, while the goal cell is highlighted in dark green

A legend is included to clarify the meaning of each color, helping to visually track how DFS explores the maze deeply along one path until it reaches the goal, without guaranteeing the shortest path.

```python
import matplotlib.pyplot as plt

def visualize_maze(maze, path):
    rows, cols = maze.shape
    plt.figure(figsize=(cols/5, rows/5))

    for r in range(rows):
        for c in range(cols):
            if maze[r, c] == 1:
                color = 'black'
            elif (r, c) in path:
                color = 'darkred'
            else:
                color = 'white'

            plt.fill_between(
                [c, c+1],
                [rows-r-1]*2,
                [rows-r]*2,
                color=color,
                edgecolor='gray'
            )

    # Start & Goal
    start_r, start_c = path[0]
    goal_r, goal_c = path[-1]

    plt.scatter(start_c+0.5, rows-start_r-0.5,
                color='darkblue', s=100, marker='s',
label='Start')
plt.scatter(goal_c+0.5, rows-goal_r-0.5,
                color='darkgreen', s=100, marker='X', label='Goal')

    plt.xticks([])
    plt.yticks([])
    plt.title("DFS Maze Pathfinding (50x50)")
    plt.legend(loc='lower left')
    plt.show()
```

A* Search

A* Search

01

- A* Search is an informed graph search algorithm that finds the shortest path by combining the actual cost from the start node and a heuristic estimate to the goal.
- Starting from the initial node, A* evaluates each node using the function: $f(n) = g(n) + h(n)$
- A* uses a priority queue to always explore the node with the lowest estimated total cost first.

02

- When applied to a maze, A* starts at the starting cell and explores neighboring cells by prioritizing those that are closer to the goal based on the heuristic (such as Manhattan distance).
- A* focuses the search toward the goal.

03

- Time Complexity: $O(n \times m)$
- Space Complexity: $O(n \times m)$

A* Search

How it works?

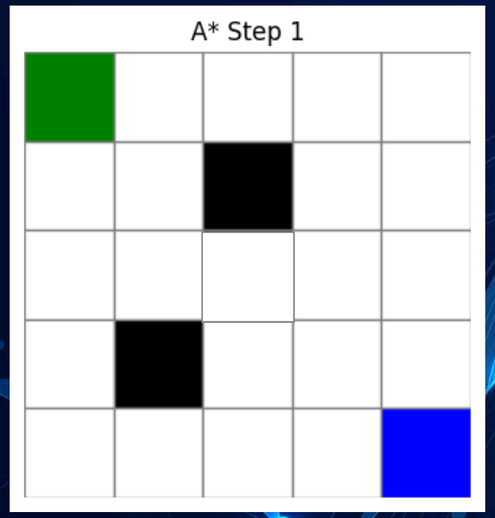
- Put the start node in the open list.

- Choose the node with the lowest f = g + h.
  - If this node is the goal, we're done.
  - If not, move it to the closed list and check its neighbors:
  - If a neighbor is not in the closed list, calculate g, h, and f.
  - If this path is better than before, update its values and add it to the open list.

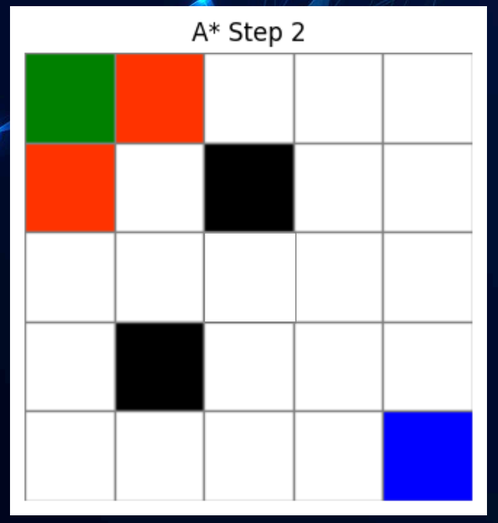
- Repeat from step 2 until the goal is reached or the open list is empty.

Example:

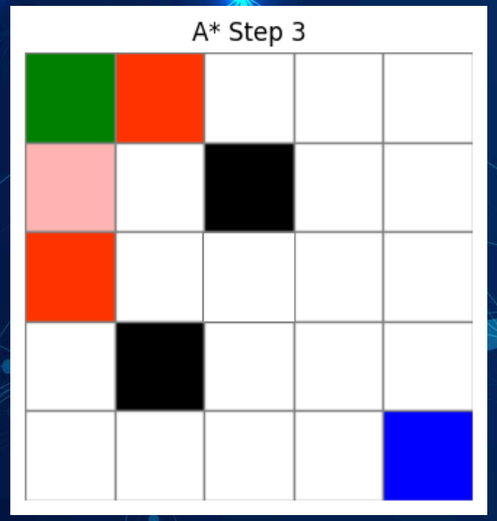
1



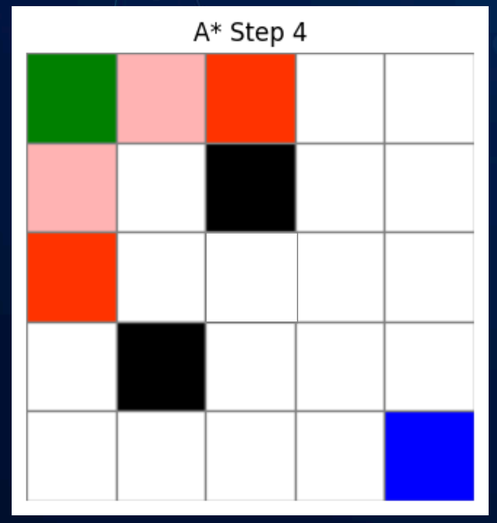
2



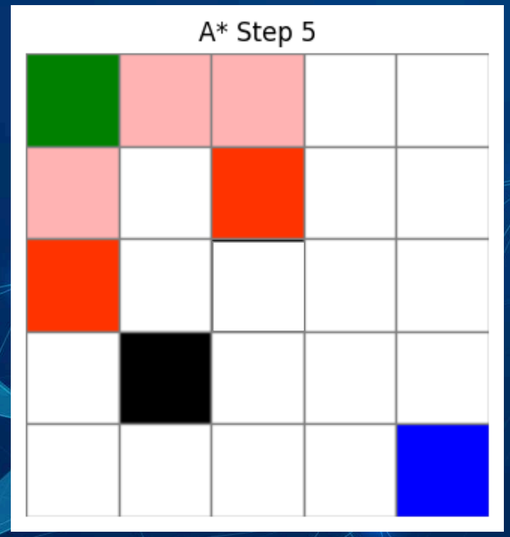
3



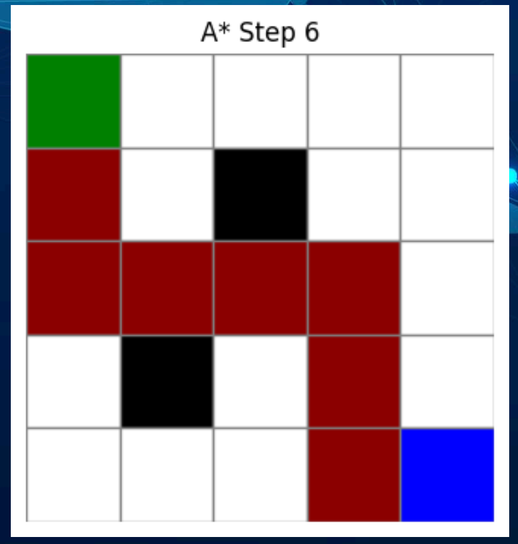
4



5



6



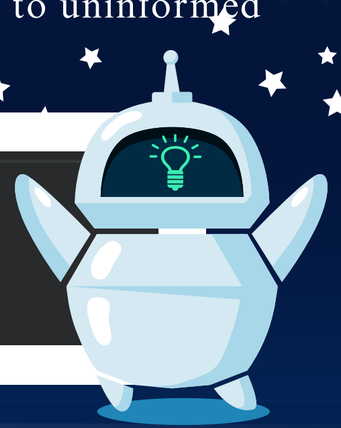
Implementation Code

```python
import heapq
import random

def generate_maze(rows=50, cols=50, wall_prob=0.3, seed=123):
    """Generate maze with seed for reproducibility"""
    random.seed(seed)
    maze = [[0 if random.random() > wall_prob else 1 for _ in range(cols)] for _ in
range(rows)]
    maze[0][0] = 0
    maze[rows-1][cols-1] = 0
    return maze

def heuristic(pos, goal):
    """Manhattan distance heuristic"""
    return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1])

def astar_maze(maze):
    rows, cols = len(maze), len(maze[0])
    start = (0, 0)
    goal = (rows-1, cols-1)
    directions = [(-1,0),(1,0),(0,-1),(0,1)]

    pq = [(heuristic(start, goal), 0, 0, 0, [(0, 0)])]
    visited = {start: 0}

    while pq:
        f_score, g_score, row, col, path = heapq.heappop(pq)

        if (row, col) == goal:
            return {
                "found": True,
                "path": path,
                "path_length": len(path),
                "nodes_explored": len(visited)
            }

        if visited.get((row, col), float('inf')) < g_score:
            continue

        for dr, dc in directions:
            nr, nc = row + dr, col + dc

            if (0 <= nr < rows and 0 <= nc < cols and maze[nr][nc] == 0):
                new_g_score = g_score + 1

                if (nr, nc) not in visited or new_g_score < visited[(nr, nc)]:
                    visited[(nr, nc)] = new_g_score
                    new_f_score = new_g_score + heuristic((nr, nc), goal)
                    new_path = path + [(nr, nc)]
                    heapq.heappush(pq, (new_f_score, new_g_score, nr, nc, new_path))

    return {
        "found": False,
        "path": [],
        "path_length": 0,
        "nodes_explored": len(visited)
    }

maze = generate_maze(rows=50, cols=50, wall_prob=0.25, seed=123)
result = astar_maze(maze)

print("Path Found:", result["found"])
print("Path Length:", result["path_length"])
print("Nodes Explored:", result["nodes_explored"])
```

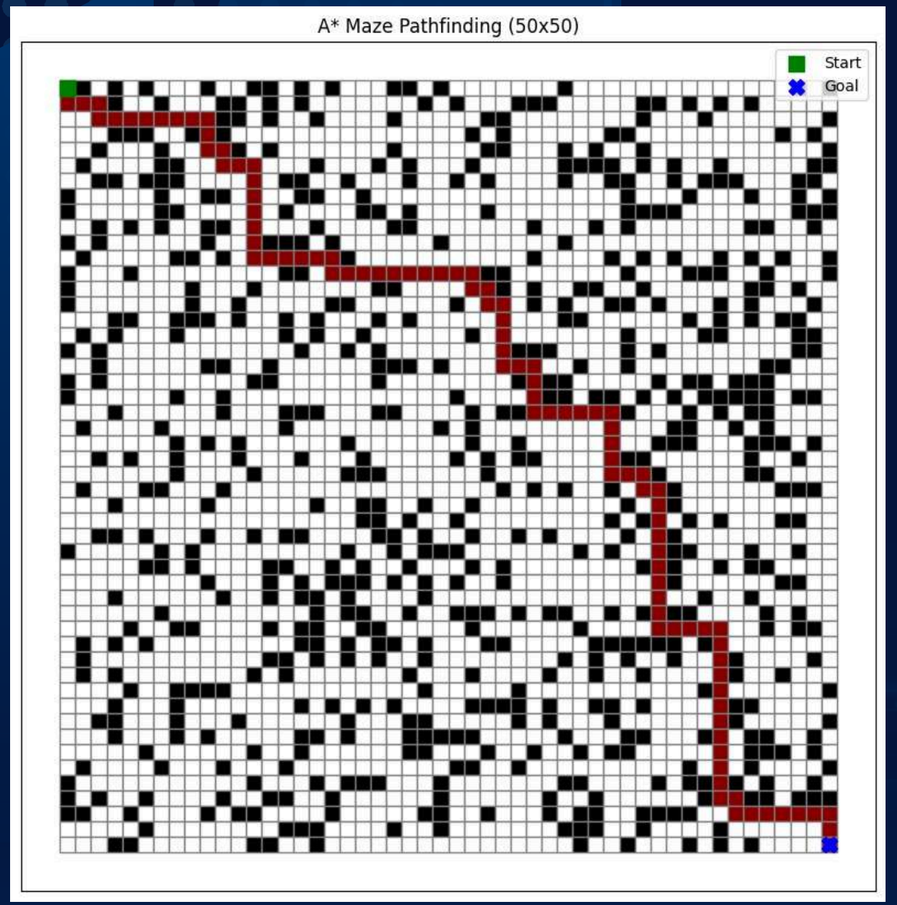
The provided code demonstrates the implementation of the A* Search algorithm to solve a maze pathfinding problem

- It first generates a random maze of size 50×50 with walls and open cells, ensuring that the start cell (top-left) and the goal cell (bottom-right) are accessible.
- A* Search starts from the initial cell (Start) and explores neighboring cells by evaluating them using the cost function: f(n) = g(n) + h(n).
- The algorithm uses a priority queue to always expand the cell with the lowest estimated total cost. Visited cells are tracked to prevent redundant exploration, while cost values are updated whenever a shorter path to a cell is found. Once the goal is reached, the algorithm reconstructs and outputs the final path from Start to Goal, the path length (number of steps), and the total number of nodes explored.
- This implementation clearly illustrates how A* efficiently finds the shortest path while reducing the number of explored nodes compared to uninformed search algorithms like BFS.

```
Path Found: True
Path Length: 99
Nodes Explored: 1039
```



Visualization Code



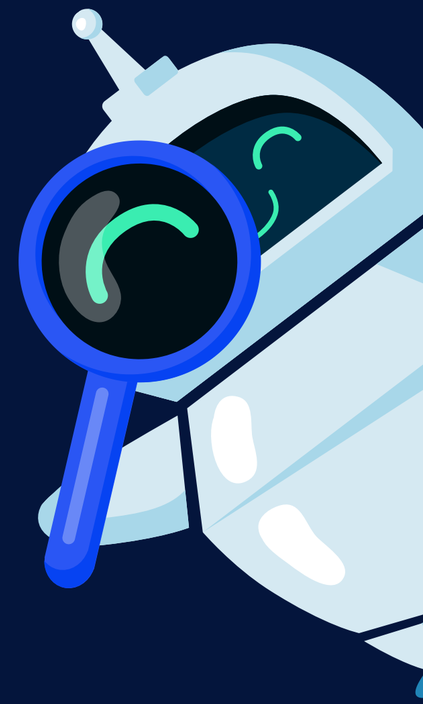
The visualization code simply draws the maze and highlights the A* search path.

Walls are shown in black, free cells in white, the optimal path found by A* in dark red, the start cell in green, and the goal cell in blue.

A legend explains what each color represents, making it easy to understand how A* navigates the maze and finds the shortest path efficiently.

```python
import matplotlib.pyplot as plt
import heapq
import random

def generate_maze(rows=50, cols=50, wall_prob=0.3, seed=123):
    """Generate maze with seed for reproducibility"""
    random.seed(seed)
    maze = [[0 if random.random() > wall_prob else 1 for _ in range(cols)] for _ in range(rows)]
    maze[0][0] = 0
    maze[rows-1][cols-1] = 0
    return maze

def heuristic(pos, goal):
    """Manhattan distance heuristic"""
    return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1])

def astar_search_maze(maze):
    if not maze or not maze[0] or maze[0][0] == 1:
        return False, []

    rows, cols = len(maze), len(maze[0])
    if maze[rows-1][cols-1] == 1:
        return False, []

    start = (0, 0)
    goal = (rows-1, cols-1)
    directions = [(-1,0),(1,0),(0,-1),(0,1)]

    pq = [(heuristic(start, goal), 0, 0, 0, [(0, 0)])]
    visited = {start: 0}

    while pq:
        f_score, g_score, row, col, path = heapq.heappop(pq)

        if (row, col) == goal:
            return True, path

        if visited.get((row, col), float('inf')) < g_score:
            continue

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc

            if (0 <= new_row < rows and 0 <= new_col < cols and
                maze[new_row][new_col] == 0):

                new_g_score = g_score + 1

                if (new_row, new_col) not in visited or new_g_score < visited[(new_row, new_col)]:
                    visited[(new_row, new_col)] = new_g_score
                    new_f_score = new_g_score + heuristic((new_row, new_col), goal)
                    new_path = path + [(new_row, new_col)]
                    heapq.heappush(pq, (new_f_score, new_g_score, new_row, new_col, new_path))

    return False, []

def visualize_maze(maze, path):
    rows, cols = len(maze), len(maze[0])
    plt.figure(figsize=(cols/5, rows/5))

    for r in range(rows):
        for c in range(cols):
            if maze[r][c] == 1:
                color = 'black'
            elif (r,c) in path:
                color = 'darkred'
            else:
                color = 'white'
            plt.fill_between([c, c+1], [rows-r-1]*2, [rows-r]*2, color=color, edgecolor='gray')

    start_r, start_c = path[0]
    goal_r, goal_c = path[-1]
    plt.scatter(start_c+0.5, rows-start_r-0.5, color='green', s=100, marker='s', label='Start')
    plt.scatter(goal_c+0.5, rows-goal_r-0.5, color='blue', s=100, marker='X', label='Goal')

    plt.xticks([])
    plt.yticks([])
    plt.title("A* Maze Pathfinding (50x50)")
    plt.legend(loc='upper right')
    plt.show()

maze = generate_maze(50, 50, wall_prob=0.25, seed=123)
result, path = astar_search_maze(maze)

print(f"Path exists: {result}")
if result:
    print(f"Shortest path length: {len(path)} steps")
    visualize_maze(maze, path)
else:
    print("No path found in this maze!")
```

GitHub Rebo & Running Codes

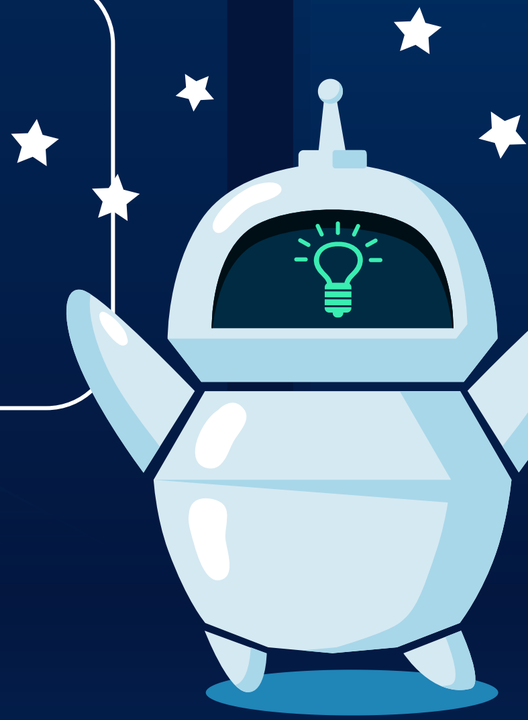


GitHub Rebo

Running Codes

Conclusion:

Maze solving can be done in many ways. Each algorithm has its own style—some are fast, some find the shortest path, and some balance both. Learning them helps us pick the best approach for any problem.



Thank You!